5
Advanced operations

PWM – pulse width modulation

In this section we will see how the output compare function can be used to create an analogue output – a simplification of the method used in the voltage inverter project. Our aim is to create a square wave output whose *mark–space ratio* we can change. The mark–space ratio is the duration of the 'logic 1' part of the wave divided by the duration of the 'logic 0' part of the wave. By controlling this ratio, we can control the output voltage, which is effectively an average of the square wave output, as shown in Figure 5.1. When using this output, you may need to add a resistor/capacitor arrangement similar to that used in the voltage inverter project, depending on the application.

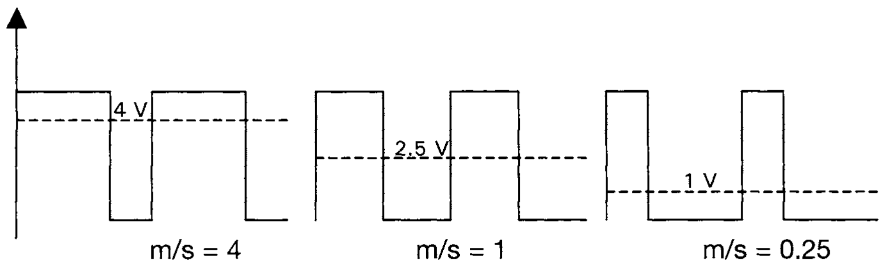


Figure 5.1

The output compare function is used to create automatic PWM, with 8-, 9-, or 10-bit resolution. By placing T/C1 in 8-bit PWM mode, for example, we force T/C1 into a mode whereby it counts up to 0xFF, and then counts back down to 0x00, and then repeats. We then set a threshold by moving a certain number into the output compare registers. When T/C1 reaches this value when counting up, it will set the OC1 output pin (PB3 on the 2313). When T/C1 reaches the value when counting back down it will clear the OC1 output pin. This creates 8-bit PWM, as illustrated in Figure 5.2.

If in 9-bit PWM mode, T/C1 will count up to 0x1FF before counting back down, giving an extra bit of resolution. Similarly, in 10-bit PWM mode, T/C1 will count up to 0x3FF and back. You are also able to invert the PWM output so

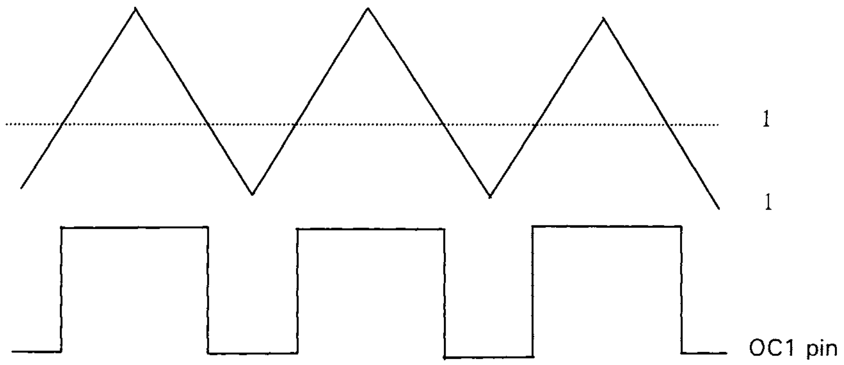


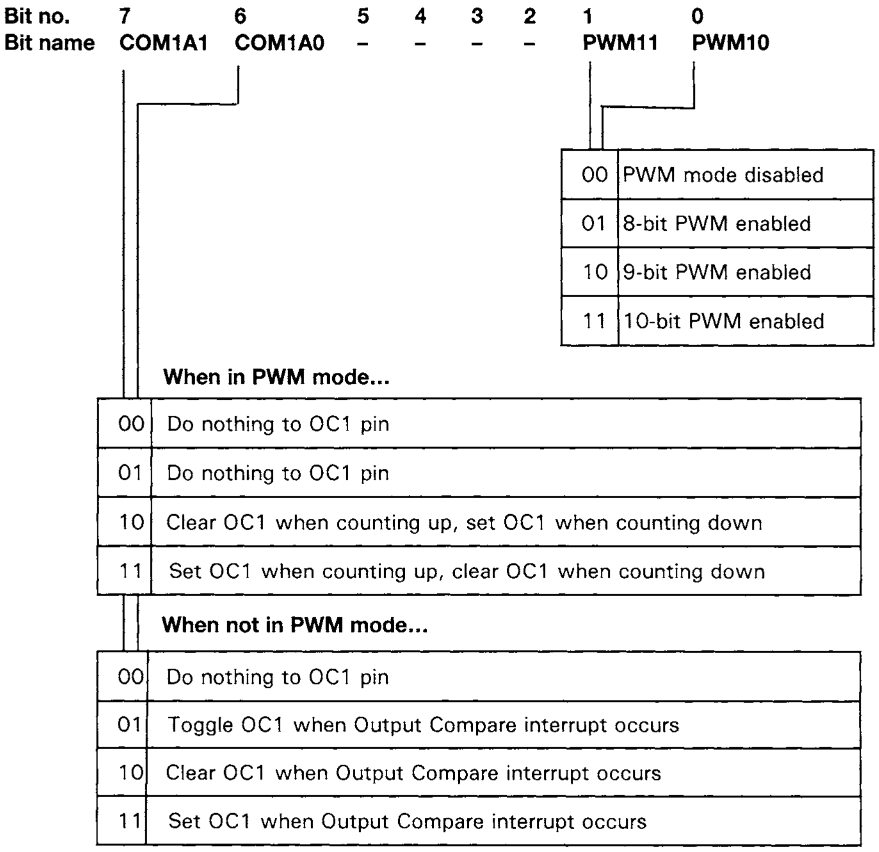
Figure 5.2

that the OC1 is *cleared* when T/C1 passes the threshold whilst counting up, and OC1 is *set* when T/C1 passes the threshold whilst counting down. The I/O register **TCCR1A** controls the PWM settings, the bit assignments are shown in Figure 5.3.

First, you will notice that you have the option, when not in PWM mode, to alter the state of the OC1 pin whenever Output Compare interrupt occurs. We could use this in the melody maker project to toggle the speaker output automatically, if we connected the speaker to OC1. You may also be wondering what happens to the T/C1 Overflow interrupt when in PWM mode (as in this case the T/C1 clearly *never* overflows). When in PWM mode, the T/C1 Overflow interrupt occurs every time T/C1 starts counting from 0x0000. Furthermore, if PWM is enabled, the OC1 is treated as an output, regardless of the state of the corresponding bit in the **DDRx** register.

There is another feature of the PWM mode which comes into effect whenever you try to change the output mark–space ratio. You would do this by changing the **OCR1AH** and **OCR1AL** registers, but unless you change them at precisely the moment at which T/C1 is at its maximum (e.g. 0x1FF for 9-bit PWM), you run the risk of a glitch appearing in your output. This glitch would take the form of a pulse whose width was in between the old and new widths. In cases where you are trying to send information encoded in the length of the pulses, this would clearly be damaging, as you would send some garbage every time you changed the pulse width. Thankfully, in PWM mode, when you try to change **OCR1AH** and **OCR1AL**, their new values are stored in a temporary location, and they are properly updated only when T/C1 is at its maximum.

<u>TCCR1A</u> – Timer/Counter 1 Control Register ($2F)



Bit no.	7	6	5	4	3	2	1	0
Bit name	COM1A1	COM1A0	-	-	-	-	PWM11	PWM10

00	PWM mode disabled
01	8-bit PWM enabled
10	9-bit PWM enabled
11	10-bit PWM enabled

When in PWM mode...

00	Do nothing to OC1 pin
01	Do nothing to OC1 pin
10	Clear OC1 when counting up, set OC1 when counting down
11	Set OC1 when counting up, clear OC1 when counting down

When not in PWM mode...

00	Do nothing to OC1 pin
01	Toggle OC1 when Output Compare interrupt occurs
10	Clear OC1 when Output Compare interrupt occurs
11	Set OC1 when Output Compare interrupt occurs

Figure 5.3

UART

'UART' is an Egyptian term that means 'the Artist's Quarter' – a place of bifurcation or division. However, UART also stands for Universal Asynchronous Receiver and Transmitter, and is a standardized method of sharing data with other devices. The UART module found on some AVR models (such as the 2313, 4433 and 8515) refers to the latter. UART involves sending 8- or 9-bit packets of data (normally a byte, or a byte plus a parity bit). This 8- or 9-bit packet is called a *character*. A parity bit is an extra bit sent along with the data byte that helps with the error checking. If there are an odd number of ones in the data byte (e.g. 0b00110100), the parity bit will be 1, if there are an even number (e.g. 0b00110011), the parity bit will be 0. This way, if a bit error occurs somewhere between sending the byte and receiving it, the parity bit will not match the data byte, the receiver will know that something has gone wrong, and it can ask for the byte to be resent. If *two* bit errors occur in one byte, the parity bit will be correct, but the probability of *two* bit errors occurring is often so small in real applications that this can be overlooked.

EXERCISE 5.1 *Challenge!* Write a short piece of code that takes the number in a register (e.g. **temp**), and works out the state of the parity bit for that register.

For transmission, the UART module takes the input character (8 or 9 bits), adds a *start bit* (a zero) at the front, and a *stop bit* (a one) to the end, to create a 10- or 11-bit sequence. This is then moved into a *shift register* which rotates the bits on to the TXD (transmission) pin, for example pin PD1on the 2313. An example is shown in Figure 5.4, and the speed at which the bits are moved on to the pin is dictated by the *baud rate* (number of bits per second) which can be controlled.

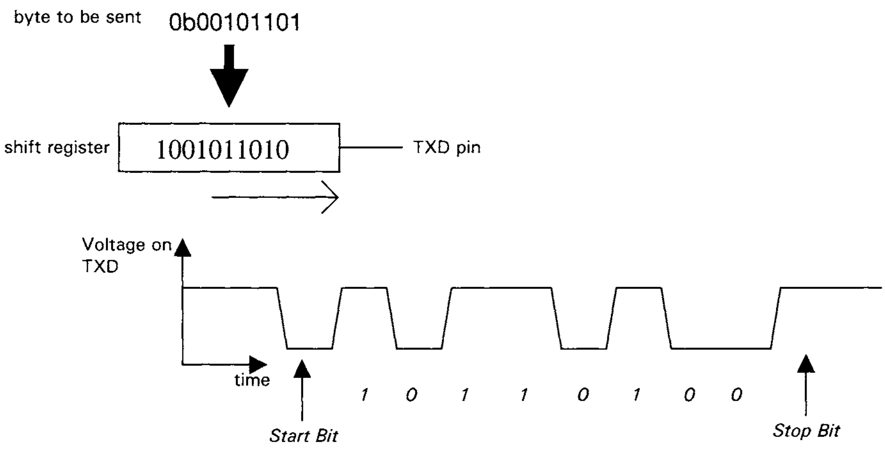


Figure 5.4

The UART module at the receiving end will be constantly checking the data line (connected to the RXD pin), which will normally be high. The receiver can actually sample the data line at *16 times* the baud rate, i.e. it can make 16 samples per bit. If it detects that the RXD pin goes low (i.e. a potential *start bit*) it waits for six samples and then makes three more samples. These should be samples 8, 9 and 10 out of the 16 for any given bit – i.e. it is sampling at the middle of the bit, allowing for slow rise and fall times on the signal. If it detects that the RXD pin is *still* low, i.e. this is *definitely* a start bit, it carries on and reads the whole byte. If the RXD is no longer low, it decides the first sample must have been noise and carries on waiting for a genuine character. If the receiver has decided that this is a genuine character, it will sample each bit three times at the middle of its pattern. If the values of the three samples taken on the same bit are not all identical, the receiver takes the majority value. Finally, when the receiver samples what it thinks should be the *stop bit*, it must read a one (on at least two of the three samples) to declare the character properly read. If it doesn't read a stop bit when it expects to, it declares the character *badly framed* and registers a *framing error*. You should check to see if a framing error has occurred before using the value you have just read into the chip.

Fortunately, all this is done for us by the UART module on the AVR chip. The UART module also brings with it four I/O registers:

UDR (UART **D**ata **R**egister, $0C) – Bits 0 to 7 of the data to be sent, or data just received
UCR (UART **C**ontrol **R**egister, $0A) – Controls settings of the UART, and contains bit 8
USR (UART **S**tatus **R**egister, $0B) – Displays status of parts of UART (e.g. interrupt flags)
UBRR (UART **B**aud **R**ate **R**egister, $09) – Sets the speed of the UART data transfer

The bit assignments for registers **UCR** and **USR** are shown in Figures 5.5 and 5.6 respectively.

Finally, **UBRR** is used to control the rate of the data transfer. Clearly, this must be the same for both the transmitting device and the receiving device. This *baud rate* is given by the following formula:

$$\text{Baud rate} = \frac{\text{CK}}{16 \times (\mathbf{UBRR} + 1)}$$

For example, if we are using a 4 MHz clock, and the number in **UBRR** is 25, the baud rate will be about 9615. There are a number of standard values for baud rates: 2400, 4800, 9600 etc., which it can be advisable to stick to, to allow compatibility of your device with others. For this reason, oscillator frequencies such as 4 MHz are not very good for UART applications, as it is impossible to choose these standard values of baud rates (try **UBRR** = 26 in the above). Much better values include 1.8432 MHz, 2.4576 MHz, 3.6864 MHz, 4.608 MHz, 7.3728 MHz, and 9.216 MHz. For the higher frequencies, make sure the AVR model you have chosen can take such a clock frequency. Taking 3.6864 MHz as an example, we can see that **UBRR** = 23 leads to a baud rate of exactly 9600.

Example 5.1 Send the value in the working register **Identity** to another UART device:

UCR - **U**ART **C**ontrol **R**egister (**$0A**)

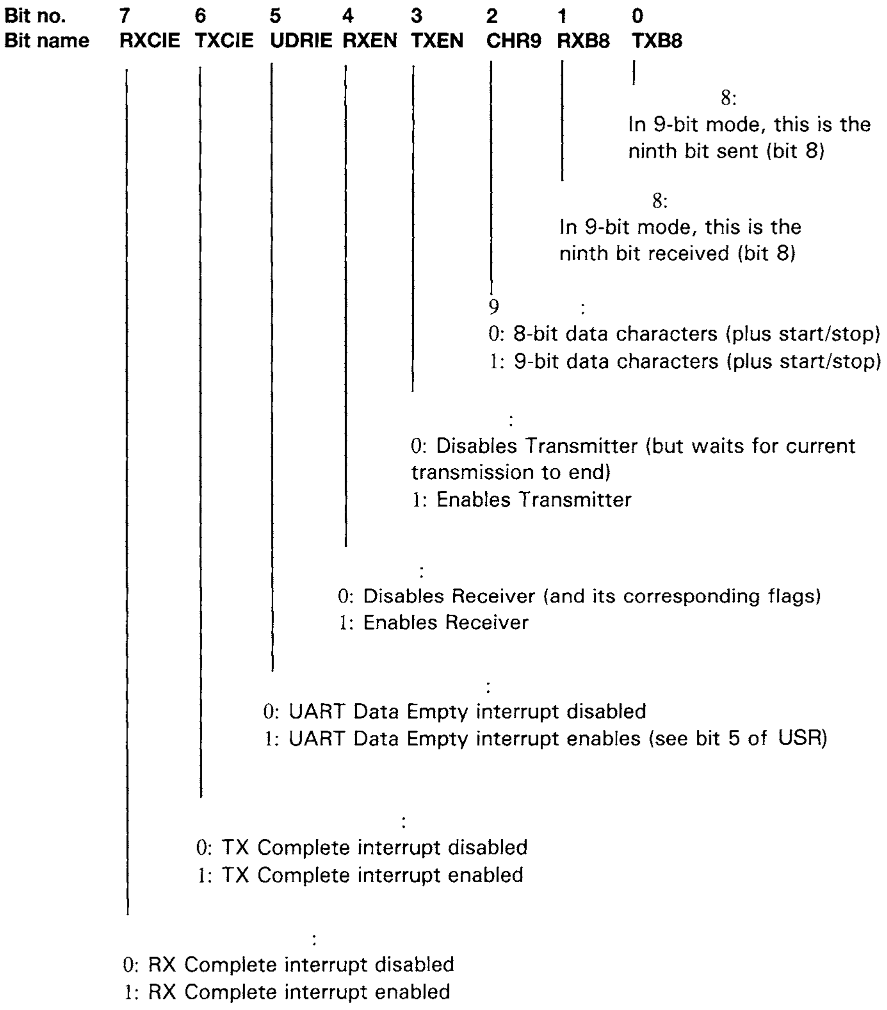

Bit no. 7 6 5 4 3 2 1 0
Bit name RXCIE TXCIE UDRIE RXEN TXEN CHR9 RXB8 TXB8
8:
In 9-bit mode, this is the ninth bit sent (bit 8)
8:
In 9-bit mode, this is the ninth bit received (bit 8)
9 :
0: 8-bit data characters (plus start/stop)
1: 9-bit data characters (plus start/stop)
:
0: Disables Transmitter (but waits for current transmission to end)
1: Enables Transmitter
:
0: Disables Receiver (and its corresponding flags)
1: Enables Receiver
:
0: UART Data Empty interrupt disabled
1: UART Data Empty interrupt enables (see bit 5 of USR)
:
0: TX Complete interrupt disabled
1: TX Complete interrupt enabled
:
0: RX Complete interrupt disabled
1: RX Complete interrupt enabled

Figure 5.5

```
ldi     temp, 0b00001000    ; enables the transmitter
out     UCR, temp           ;
out     UDR, Identity       ; sends value
```

If we wished to send another piece of data, we would have to wait for the **UDRE**

<u>**USR**</u> – **U**ART **S**tatus **R**egister (**$0B**)

Bit no.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Bit name	**RXC**	**TXC**	**UDRE**	**FE**	**OR**	**–**	**–**	**–**

:

0: has been successfully transferred to shift register
1: has been overwritten before byte was moved into shift register

:

0: No framing error (stop bit is ok)
1: Framing error detected (bad stop bit)

:

0: The byte in UDR hasn't yet been emptied into the shift register
1: has been emptied into the shift register

:

Set when a character has been transmitted, and there is no new data in UDR. If the interrupt is enabled, this bit is automatically cleared. If not, it must be cleared by *setting* the bit.

:

Set when a character has been received and stored in UDR. If the interrupt is enabled, must be read to clear this bit.

Figure 5.6

bit in **USR** to tell us that the byte has been moved into the shift register, and **UDR** is ready for a new byte.

You can use UART to communicate with the RS232 port on your PC. The simplest way to send bytes through your PC's serial port is through a program that comes with Microsoft® Windows® called HyperTerminal (Start Menu → Programs → Accessories → Communications). You can create a connection with your serial port (e.g. COM1), choose a baud rate, number of bits, parity setting etc. When HyperTerminal connects to the serial port, whatever character you type is sent (as ASCII) through the serial port. If you have a development board, such as the STK500, there is an RS232 socket that you can connect directly to the RXD and TXD pins. If you do not have such a development board, you will have to wire up the correct pins to RXD and TXD, and also make sure the voltage (which could be anywhere between 3 and 12 V), is regulated to a safe voltage (like 5 V). Figure 5.7 shows how to wire up the pins on a 9-pin RS232 socket to allow direct communication with the AVR. Some of the other pins are *handshaking pins*, which can be bypassed by connecting them together as shown.

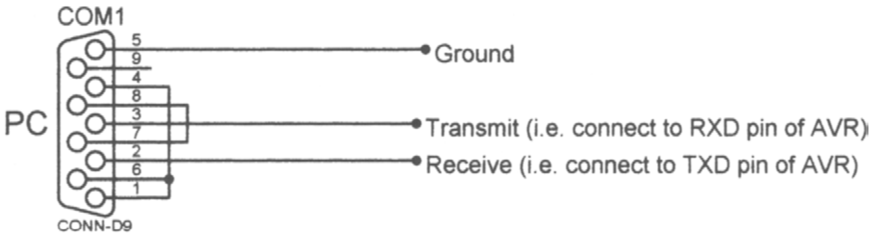

COM1
PC
5
9
4
8
3
7
2
6
1
CONN-D9
Ground
Transmit (i.e. connect to RXD pin of AVR)
Receive (i.e. connect to TXD pin of AVR)

Figure 5.7

Program O: keyboard converter

- UART
- Sounds
- Seven segment displays
- Output compare

We can use HyperTerminal to send characters to our melody maker project, via the UART module. We can effectively convert our computer keyboard into a musical keyboard by assigning note frequencies to the different characters. For example, when I press the letter 'a' when HyperTerminal is connected to the AVR, it will send 'a' to the UART module. This can then trigger an interrupt, convert the ASCII code for 'a' into the frequency for a 'C' note. I have arranged my keys on the keyboard so that they resemble how they are arranged on a piano, but you may find you can fit more notes if you arrange them differently. Figure 5.8 shows my arrangement.

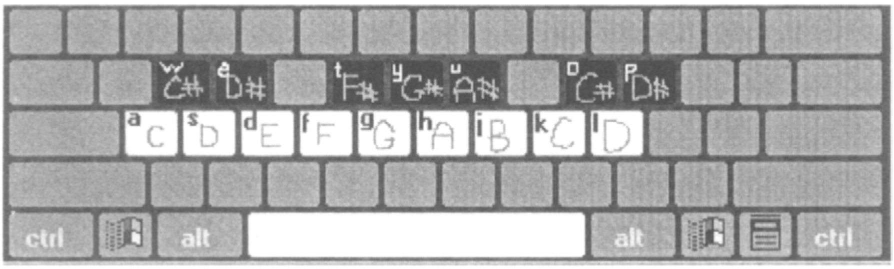


Figure 5.8

We will also use a seven segment display to show which note is being played; this can help overcome any user confusion over how the letters on the computer keyboard correspond to the musical notes. There will be a separate LED to show the sharp symbol (#). The circuit diagram is shown in Figure 5.9, and the flowchart in Figure 5.10.

In the Init section, set up inputs and outputs and set OC1 to toggle with every output compare (this handles the speaker output for us, so there is no need to write a routine for the Output Compare interrupt). Make all other timer settings the same as in the melody maker, choose a baud rate of 9600, and enable the UART receiver and the UART Receive Complete interrupt.

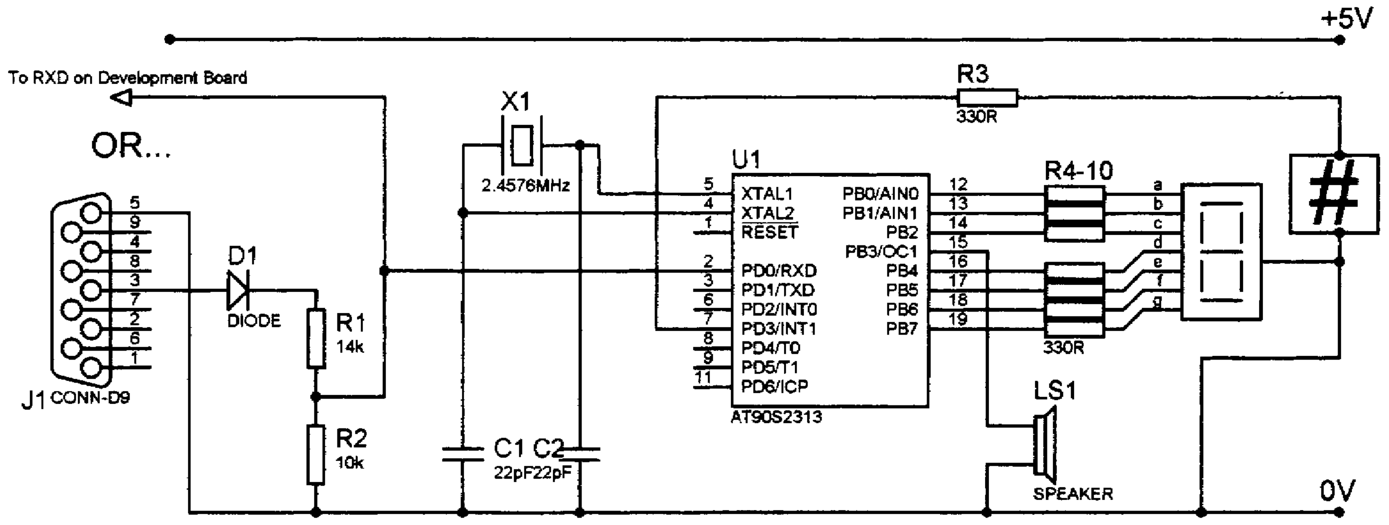
Again, the main body of the program is just a constant loop to **Start**. The UART Receive Complete interrupt tells us that some new data has been received on the line, which we should convert to a frequency and then change **OCR1AH** and **OCR1AL** accordingly. The beginning of the interrupt routine should therefore read **UDR** into ZL. The ASCII conversion table is shown in Appendix G. I will only use letters a–z, all lower case, which correspond to 0x61 to 0x7A in ASCII, so subtract 0x61 from ZL to get a number between 0 and 25. If ZL is more than 25, an inappropriate key is being pressed, so move 26 into it, this ensures no matter what character we read, the program will stay within the look-up table we are about to write. Now multiply ZL by two to make it a word address. We wish to read the program memory into R0, using the **lpm** instruction, and then copy R0 into **OCR1AH**, and **OCR1AL**. We can do this directly (i.e. without having to play with octaves etc., so we don't need **NoteH** and **NoteL**). However, when doing this directly, we have to remember the golden rule – you must write the higher byte first. There are two ways of doing this. First, arrange the data in the look-up table so that the higher byte actually comes first. For example, if I wished the number 0x1E84 to be the code for a 'C' note, the top of my look-up table would be:

```
.dw         0x841E
```

This is a little confusing, and an easier way is to start by pointing ZL to the higher byte. In other words, if the table starts at byte address 26 in the program memory, add 27 to ZL instead of 26, to point ZL to the higher bytes. Then to read the lower byte, *decrement* ZL.

EXERCISE 5.2 *Challenge!* Write the first *12* lines of the UART Receive Complete interrupt section which use the data received by the UART module to write new values for **OCR1AH** and **OCR1AL**.

For the display we have another look-up table, below the first, starting at word address 43. We can simply add 60 (30 × 2) to ZL to point to the second look-up table. This holds the seven segment codes for the note letters. Bit 3 will be used to light up the # (sharp) LED.



+5V
To RXD on Development Board
OR...
X1
2.4576MHz
R3
330R
U1
XTAL1
XTAL2
RESET
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/T0
PD5/T1
PD6/ICP
PB0/AIN0
PB1/AIN1
PB2
PB3/OC1
PB4
PB5
PB6
PB7
AT90S2313
R4-10
330R
a
b
c
d
e
f
g
LS1
SPEAKER
J1 CONN-D9
D1
DIODE
R1
14k
R2
10k
C1 C2
22pF22pF
0V

Figure 5.9

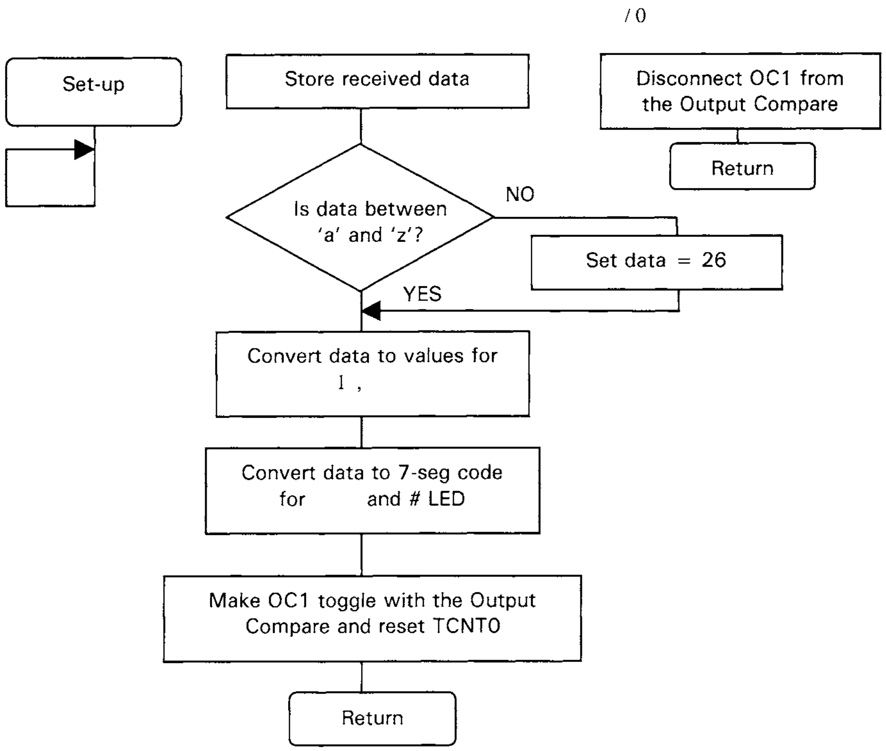

/0
Set-up
Store received data
Is data between 'a' and 'z'?
NO
YES
Set data = 26
Convert data to values for
1 ,
Convert data to 7-seg code for and # LED
Make OC1 toggle with the Output Compare and reset TCNT0
Return
Disconnect OC1 from the Output Compare
Return

Figure 5.10

EXERCISE 5.3 What *six* lines point ZL to the second look-up table, read the value, and output it to **PortB**? They should then mask all of R0 (which contains the value read from the table) except bit 3, and move the result into **PortD**, to take care of the # LED. As you cannot use the **andi** instruction on R0–R15, you will have to copy R0 into **temp**.

EXERCISE 5.4 What *five* lines will set the OC1 pin to toggle with every Output Compare interrupt, reset T/C0 and return?

EXERCISE 5.5 What *three* lines make up the T/C0 Overflow interrupt, which should disconnect the OC1 pin from the Output Compare interrupt and return?

This program is quite fun to play around with, but you may find the keyboard's *repeat delay* a nuisance. You can try to minimize this in the Control Panel, or perhaps lengthen the minimum note to try to overcome it. If you move the frequencies produced out of the audible range, this project can be developed into more sinister applications – perhaps you could use it for espionage purposes ...?

Another UART project you may wish to make would be to build upon the palindrome detector designed in Chapter 3, and interface it with a computer via its serial port. The use of the Receive Complete interrupt would simplify the program considerably.

Serial peripheral interface (SPI)

The UART described in the previous section has a few drawbacks. For a start it is only *half duplex* (also called *simplex*) – this means you can send data in only one direction on one line. Connecting the TXD pin on one device connected to the RXD pin of another supports data transfer in one direction only, namely TXD to RXD. SPI offers *full duplex* – the ability to send data in both directions at the same time. It is also a *synchronous* mode of transfer – this means all the relevant devices are also connected to a common clock, so that they can all be in synch, and operate at a higher speed.

Sending information through the SPI module is just as straightforward as with UART. Any number of SPI devices can be connected together; however, one device is called the Master, and the other devices are Slaves. The Master can talk to the Slaves, and the Slaves can talk to the Master, but the Slaves cannot talk to each other. The Master provides the clock that synchronizes the connection, and it decides when it is going to talk to the Slave, and when the Slave can talk to it. Figure 5.11 shows an arrangement with one Master and two Slaves.

When you move a number into the SPI data register of the Master device, it will immediately start a clock signal on the SCK pin (SPI **C**loc**k**), and begin shifting the data out on the MOSI pin (**M**aster **O**ut, **S**lave **I**n) to the Slaves on

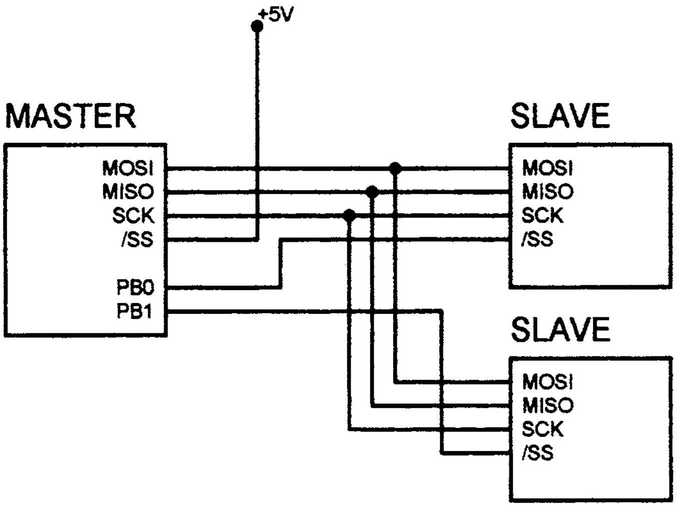


Figure 5.11

their MOSI pins. The Slave will receive the data only if it has been chosen by the Master, i.e. if its $\overline{SS}$ pin is high. Therefore, using any two output pins (PB0 and PB1 in the example in Figure 5.11), the Master can choose which of the Slaves it wants to talk to. As the Master sends its data to the Slave on the MOSI pin, the Slave immediately begins sending the contents of its data register to the Master on their MISO (**M**aster **I**n, **S**lave **O**ut) pins. The two 8-bit shift registers on Master and Slave behave like one big, circular 16-bit shift register – as bits shift off Master onto Slave, bits shift off the Slave and into the Master. You can configure the $\overline{SS}$ pin on the Master as an output, and use it as a general output. If you make it an input, however, you must tie it to V_{CC}, as shown. If the Master's $\overline{SS}$ pin is pulled low, it assumes some other Master wants to enslave it, and will turn into a Slave! This allows some hierarchy between Masters in a complex SPI system. The I/O registers involved with SPI are:

SPDR (**SP**I **D**ata **R**egister, $0F) – Data to be sent, or data just received
SPCR (**SP**I **C**ontrol **R**egister, $0D) – Controls settings of the SPI
SPSR (**SP**I **S**tatus **R**egister, $0E) – Displays status of parts of SPI (e.g. interrupt flags)

SPDR is the data register into which you should move the byte to be sent to the other device, and holds the received byte after the transmission is finished. You must wait for the current transmission to finish before writing the next byte to be sent to **SPDR**. When reading the received byte, you have slightly longer to read it. You can read the received byte while the next transmission is in progress, but once this next byte is completely received, the old received byte is overwritten. You therefore have until the next transaction *completes* to read the received data.

The **SPSR** contains two flags. Bit 6 is the *write collision flag*, which is set when **SPDR** is written to before the current transmission is finished. Bit 7 is the *SPI interrupt flag*, which is set when an SPI transmission completes.

An example project you may wish to consider attempting could be an electronic chess game involving two AVR units which communicate using an SPI link. The users at either end can input their move into their unit, which will then send the move to the other unit. The game can be stored on the EEPROM (thus allowing games to continue after power has been removed and the units separated). Sixty-four bytes are required, as each square on the board can be assigned a space in the EEPROM. The number in the EEPROM indicates which piece is on that space. For example 00 could mean empty. 01 = black pawn, 02 = black knight etc., 81 = white pawn, 82 = white knight etc. The allowed moves would involve adding or subtracting numbers to a particular piece's position. For example, allowed moves for bishops are at the basic level adding or subtracting multiples of 9 or 7. Figure 5.13 should help you picture this. However, tests will be needed to ensure the piece doesn't travel through another, or off the board.

<u>**SPCR**</u> – **SPI C**ontrol **R**egister (**$0D**)

Bit no.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Bit name	**SPIE**	**SPE**	**DORD**	**MSTR**	**CPOL**	**CPHA**	**SPR1**	**SPR0**

00	SCK speed is CK/4
01	SCK speed is CK/16
10	SCK speed is CK/64
11	SCK speed is CK/128

:
0: Trigger on rising edge of SCK
1: Trigger on falling edge of SCK

:
0: SCK pin low when idle
1: SCK pin high when idle

/ :
0: Slave mode
1: Master mode

:
0: MSB of data word transmitted first
1: LSB of data word transmitted first

:
0: SPI disabled
1: SPI Enabled. MOSI, MISO, SCK and SS pins enabled

:
0: SPI interrupt disabled
1: SPI interrupt enabled

Figure 5.12

The moves could be entered in standard chess notation (e.g. Be2 = Bishop to the E2 square), or with the help of a more visual display which resembles the board. This project is left as an exercise for the chess enthusiasts, but I would be interested in seeing your solutions (my email address is given in Appendix I).

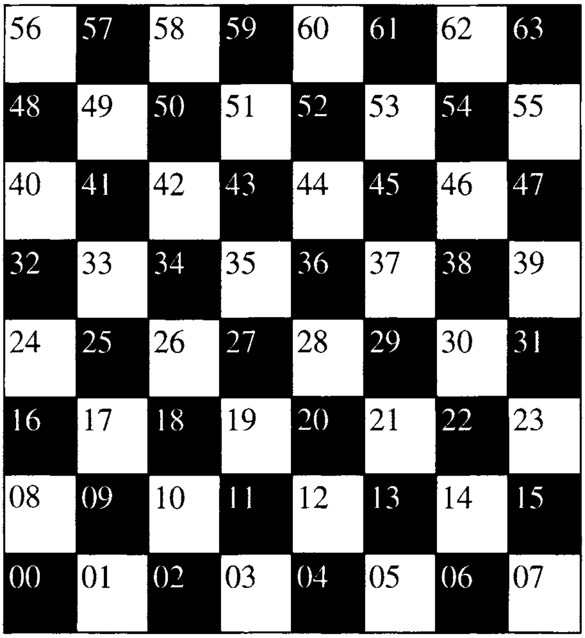


Figure 5.13

Both UART and SPI can be implemented on chips without these custom modules, entirely with software. For more information on these, you can check out Claus Kühnel's book listed in Appendix I, but my advice would be simply to use a chip that has the hardware you require.

Tiny15's eccentric timer 1

As a brief aside, it is worth noting that the Tiny15 has an 8-bit T/C1, and a few other eccentricities that make it different from the norm. Whereas on other chips, T/C0 and T/C1 can count up at no more than CK, the clock speed at which instructions are performed, the T/C1 on the Tiny15 can actually count up *faster* than CK. It can be set to count at 16CK, 8CK, 4CK or 2CK, as well as CK, and also at a larger range of fractions of CK, as shown in the Tiny15's bit assignment of **TCCR1**, the **T/C1** Control **R**egister (Figure 5.14). The reason it can count higher than CK is that it has access to a high-speed clock (called PCK) that runs 16 times faster than CK; values such as 8CK and 4CK are obtained by prescaling this high-speed clock.

As T/C1 is only 8 bit, the PWM is 8 bit. Rather than counting up and down in PWM mode, T/C1 is always counting up, and will change the state of the OC1 pin when it reaches the top. The top value of T/C1 is given by the **OCR1B**

TCCRI – **T/CI C**ontrol **R**egister (**$30**) on the Tiny15

Bit no.	7	6	5	4	3	2	1	0
Bit name	CTC1	PWM1	COM1A1	COM1A0	ADIE	ADPS2	ADPS1	ADPS0

0000	STOP! T/C1 is stopped
0001	T/C1 counts at 16 x CK
0010	T/C1 counts at 8 x CK
0011	T/C1 counts at 4 x CK
0100	TC/1 counts at 12 x CK
0101	T/C1 counts at CK
0110	T/C1 counts at CK/2
0111	T/C1 counts at CK/4
1000	T/C1 counts at CK/8
1001	T/C1 counts at CK/16
1010	T/C1 counts at CK/32
1011	T/C1 counts at CK/64
1100	T/C1 counts at CK/128
1101	T/C1 counts at CK/256
1110	T/C1 counts at CK/512
1111	T/C1 counts at CK/1024

... ...

00	Do nothing to OC1 pin
01	Do nothing to OC1 pin
10	Clear OC1 when compare match, set on T/C1 overflow
11	Set OC1 when compare match, clear on T/C1 overflow

... ...

00	Do nothing to OC1 pin
01	Toggle OC1 when Output Compare interrupt occurs
10	Clear OC1 when Output Compare interrupt occurs
11	Set OC1 when Output Compare interrupt occurs

:
0: PWM disabled
1: PWM enabled (8-bit)

/ I :
0: Doesn't reset T/C1 on Compare Match
1: T/C1 is reset to $00 on Compare Match

Figure 5.14

I/O register. The PWM is glitch free, as before, so updates to **OCR1A** occur only when T/C1 reaches the top value, as shown in Figure 5.15.

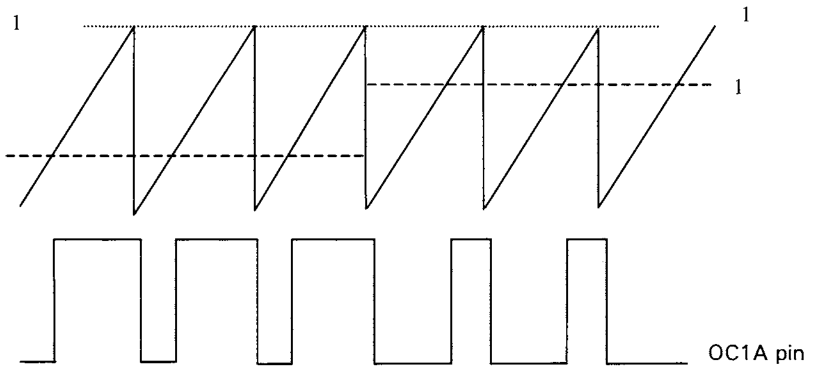


Figure 5.15

As if this wasn't enough, there's another I/O register thrown in, with the mysterious title of **S**pecial **F**unction **IO R**egister: **SFIOR** ($2C). This register allows you to *reset the prescaler* of either of timer/counters. What on earth does this mean? Let's look at how the prescaler works. Essentially, the prescaler is a 10-bit register that counts up at CK. When T/C0, for example, is 'prescaled at CK/2' it counts with bit 0 of the prescaler. If it is 'prescaled at CK/64', it counts with bit 5 of the prescaler etc. This is illustrated in Figure 5.16.

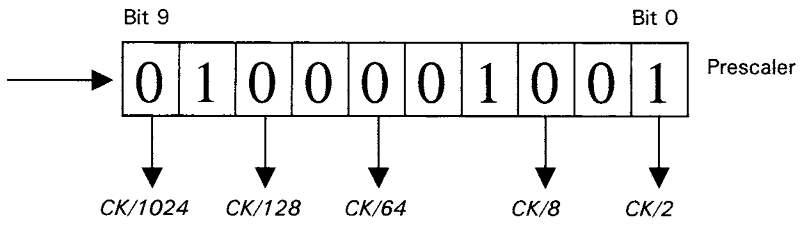


Figure 5.16

When you reset the prescaler, you wipe its value to 0, ensuring a more accurate count. Say you wished to set your T/C0 to count at CK/1024. In *steady state* operation it will be perfectly accurate, but for that very first count, we don't know that the number in the prescaler doesn't happen to be 1023, and so the first count will come a lot sooner than expected. To reset the prescaler for T/C0, just set bit 0 of **SFIOR** (the bit will then clear itself). To reset the prescaler for T/C1, set bit 1 of **SFIOR**. Finally, with bit 2 of **SFIOR**, we are able to force a change on the OC1A pin, according to the settings in bits 4 and 5 of **TCCR1**. In other words, we 'fool' the pin into thinking there has been an Output

Compare Match; however there is *no* interrupt generated, and T/C1 will *not* reset.

Although at the time of publication, the Tiny15 was the only model with this type of T/C1, we can expect that other models of AVR will emerge with a similar T/C1.

Shrtcts

There are a number of ways to trim down your program into a slender and seductive beauty. One of the easiest ways is to use the **.macro** assembler directive. This allows you to, in effect, create your own instructions.

Example 5.1 At the top of your program ...

```
.macro      nopnop                  ; the name of this macro is nopnop
            rjmp   PC+1
.endmacro
```

Then, in the rest of your program, you can write the instruction **nopnop**, and the assembler will interpret this as **rjmp PC+1**. Why have I called this **nopnop**? Jumping to the next line with the **rjmp** instruction wastes *two* clock cycles, as the **rjmp** instruction takes twice as long as most instructions. Writing **rjmp PC+1** is therefore equivalent to writing two **nop**s, but only takes up one instruction. Macros can also be given operands, which are referred to as @0, @1 etc.

Example 5.2

```
.macro      multiply                ; the name of this macro is multiply
            mov    temp, @0         ;
            clr    @0               , wipes answer register
            tst    @1               ; tests multplier
            breq   PC+4             ;
            add    @0, temp         ; adds multiplicand to itself
            dec    @1               ;
            rjmp   PC-4             ; repeats
.endmacro
```

In the program, if we wanted to multiply the number in **Seconds** by the number in **Counter**, we could simply write:

```
            multiply     Seconds, Counter
```

Note that we can use labels in the macro, these will immediately be translated as relative jumps, and so there will be no risk of label duplication should the macro be used more than once in the program.

EXERCISE 5.6 Create a macro called **skeq** which skips the next instruction if the zero flag is set.

EXERCISE 5.7 Create a macro called **HiWait** which will wait until a bit in an I/O register goes high.

It is important to clarify in your mind the distinction between subroutines and macros. Macros are simply ways of abbreviating longer or less pretty pieces of code into neat one-word actions. The assembler will expand these out, so your program will end up just as long (but you will never see the expanded version). Using subroutines will actually make your program shorter (i.e. take up less space in the program memory), BUT may well take longer to run. The **rcall** instruction takes three clock cycles, and the **ret** instruction four clock cycles, so subroutines are literally a waste of time for really short shortcuts.

A Mega summary

Covering the cornucopia of new functions found on the MegaAVR range is not one of the aims of this book. It is worth, however, giving a brief introduction so that you can at least decide whether it's worth learning more about them. First, they offer more of what you've seen so far: more timers, more PWM, more ADCs, more I/O pins, more memory and more instructions.

The new instructions fall into three categories. There are a few new instructions introduced along with an *on-chip multiplier* – specially built hardware which performs multiplication in two clock cycles. The **mul** instruction is used to multiply two registers together. Other multiply instructions (signed/unsigned/fractional etc.) are also available. The **call** and **jmp** instructions are direct calls and jumps respectively. The only difference to the user is the ability to jump to, or call, *any* part of the program, though you probably won't experience this limitation on non-Mega AVRs unless you write really large programs. The new instructions also include additions to the memory access instructions, most notable is the **stm** instruction. This stores the *word* spread over R0 and R1 into the program memory. This allows the program to write to itself!

Another particularly useful feature available on most new AVRs is the *JTAG interface*. This is a standard that has been developed to facilitate debugging. It is a way for the AVR to send the entire contents of its registers (I/O, working registers, SRAM) to a PC, so that you can see what's going on inside it as it runs in your circuit board.

Final program P: computer controlled robot

- Serial communication
- PWM to drive a motor
- Seven segment display to display messages

A computer controlled robot has been chosen as a fun project which ties together some of the topics discussed in the book. The project that will be developed will be a skeleton, around which a semi-intelligent robot can be based. We can send commands to the robot through the serial port on the computer to the UART module on the AVR. Motor speed can be controlled through the use of PWM, and a seven segment display will be used to show messages, and allow the robot to 'talk'. The use of EEPROM to store moves and the application of the music modules are some basic enhancements that could be added on. Sensors could be placed on the robot, and it could send information back to the computer regarding the states of these sensors. More sophisticated software on the computer end, which would make the robot behave like a state machine and respond to various inputs, would be a more interesting development, but this goes beyond the scope of this book. The circuit diagram of the basic robot is shown in Figure 5.17.

Both motors are driven from the OC1 pin, which is the output of the PWM. To allow the robot to turn, the left motor can be turned off by setting the PD2 pin. This means it can turn in one direction only, but still gives it plenty of freedom. A larger AVR, such as the 8515, has two PWM outputs, on OC1A and OC1B pins. This means the motors can be driven independently.

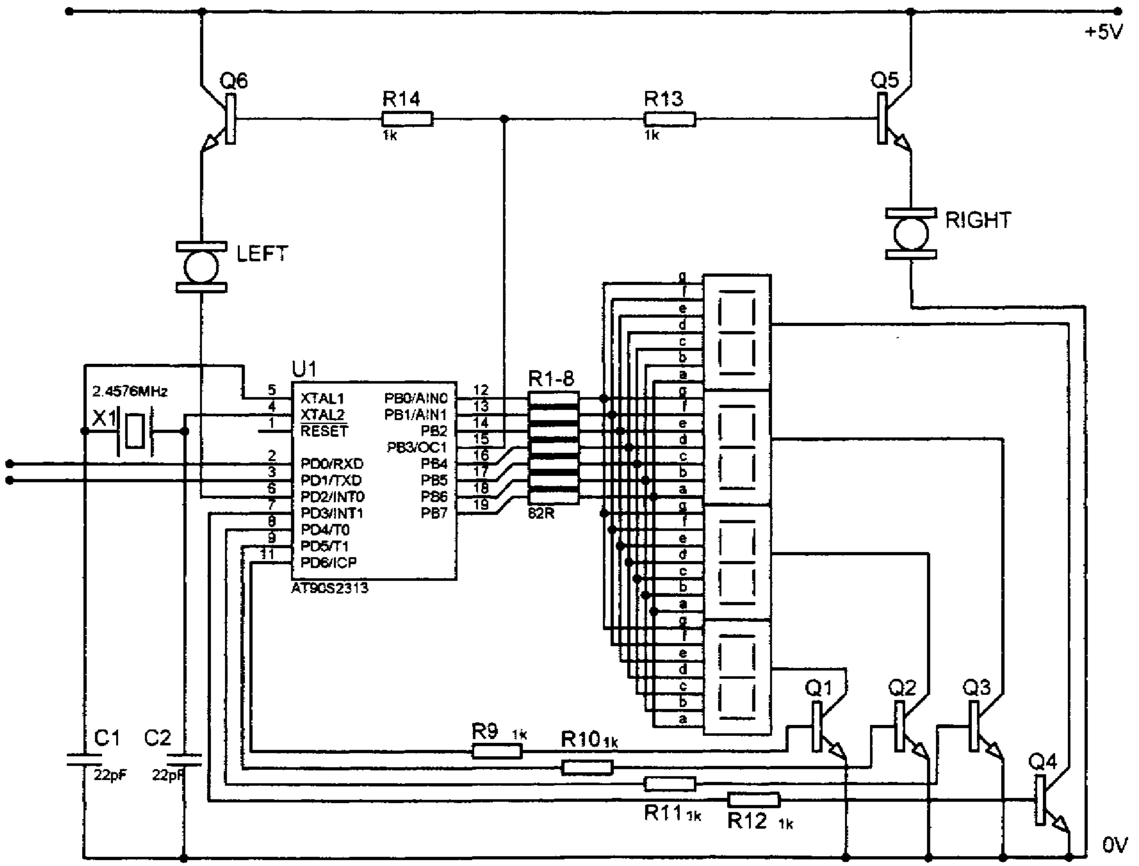
The commands we can send the robot are shown in Table 5.1.

Table 5.1

Letter	ASCII	Function	Message to PC
g	0x67	Go/Stop	'Go' or 'Stop'
t	0x74	Begin turning or end turning (stop/start left motor)	'Turning'
+	0x2B	Speed up	'Speeding up'
-	0x2D	Slow down	'Slowing down'
s	0x73	Change speed (followed by two-digit number, e.g. **s25**)	'Speed set to ...'
[	0x5B	Begin message (to be displayed on seven segment displays)	<message>
]	0x5D	End message	

All other inputs will be ignored. The robot will send the computer back confirmations of each action. For example, if it is sent a '**t**', it will reply with 'Turning'. Not all letters can be displayed on the seven segment displays – to be able to display any letter we need a more complex display (e.g. a 14 segment display). As it is, we are unable to display letters k, m, q, v, w and x.

The structure of the program is very straightforward, and entirely interrupt driven. If a receive interrupt occurs, the program identifies the character received and responds accordingly. To simplify the **Display** subroutine, we can make this



+5V
Q6
R14
1k
R13
1k
Q5
RIGHT
LEFT
U1
2.4576MHz
X1
XTAL1
XTAL2
RESET
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/T0
PD5/T1
PD6/ICP
PB0/AIN0
PB1/AIN1
PB2
PB3/OC1
PB4
PB5
PB6
PB7
AT90S2313
R1-8
82R
g f e d c b a
Q1
Q2
Q3
Q4
R9 1k
R10 1k
R11 1k
R12 1k
C1 22pF
C2 22pF
0V

Figure 5.17

driven by T/C0, such that every time T/C0 overflows, the **Display** subroutine is called. This not only removes the burden on us of remembering to call it regularly, but also means we can remove the counter register that allows the entire subroutine to be executed only once every 50 visits. We must therefore configure T/C0 so that it overflows sufficiently often. The refresh rate should be more than 25 times a second and, bearing in mind there are four displays, this means the **Display** subroutine should be called at least 100 times a second. As T/C0 overflows after 256 counts, this means a minimum T/C0 rate of 25.6 kHz. If we are using a 2.4576 MHz crystal, this represents prescaling of CK/64.

In the Init section, configure the inputs and outputs, and T/C0. Set up T/C1 to count at CK, set OC1 to clear when T/C1 passes the threshold counting up, and set when T/C1 passes it coming down (this means the higher the number in **OC1AH/L**, the faster the speed of the motor). Disable PWM for the time being (8-bit PWM will be enabled when a '**g**' is received from the computer). Don't forget to set up the stack pointer I/O registers. On the 2313 this is just **SPL**, and which you should load with 'RAMEND'. Enable the Receive Complete UART Interrupt, and enable the Receive Mode. Set the UART baud rate to 9600, and enable the global interrupt bit.

Adjust the **Display** subroutine from previous projects to include *four* displays. The seven segment code to be displayed will be stored in registers **R21–24**. Note that as these will hold the seven segment code, their values can be moved directly into PortB.

EXERCISE 5.8 Make the necessary changes to create a **Display** subroutine for this program.

The Receive Complete Interrupt should first test to see if what is being sent is to be taken as a command, or as part of a text message. The T bit will be used to indicate which interpretation is appropriate (i.e. the start message command '[' will set the T bit, and the end message command ']' will clear it. It should also be cleared in the Init section. The Receive Complete Interrupt section should start by testing for an end message symbol, and jump to **EndMessage** if it is received. The next test should be the T bit, if it is set we should branch to **Message**. The other symbols (g, t, s, +, –) can be tested in any order, though it is simplest to put the test for '[' at the end. If it is '[', the T bit should be set. Any other symbol should be ignored.

The **Turning** section should toggle the state of the PD2 pin (which controls the left motor). The receive mode should then be disabled, and the transmit mode enabled. Move the ASCII code for a 'T' into **temp**, and then call a subroutine called **Send**. This subroutine will take the number in **temp** and send it through the UART module; we will write the subroutine later. Repeat the above for the rest of the letters. We also need to send a *new line* (also called *line feed*) and *carriage return* symbol, so that each message sent to the PC appears on a new line. These symbols are 0x0A and 0x0D respectively, but these will be common to all messages, so at this point (after sending the 'g'), just branch to **EndMessage**, which will do the rest.

EndMessage will clear the T bit, send 0x0A and 0x0D to the PC, and then disable the transmit mode and enable the receive mode.

The **Send** subroutine should put the contents of **temp** into the **UDR**, and then enter a loop in which it constantly checks the transmit complete flag (the TXC bit in **USR**). You must not write to UDR in this loop (i.e. loop to **Send+1**, and not to **Send)**, because this resets the TXC flag, which means you will stay in the loop forever. After the TXC flag goes high, you must reset it by *setting* it, and then return.

The **SpeedUp** section will read in the number currently in OCR1AL, and add 10 to it. If the carry flag is set, the number should be capped at 0xFF, and then moved back to OCR1AL. Note that you *cannot* use the following:

```
        subi    temp, -10
```

This really adds 246 to **temp**, which will almost invariably set the carry flag. You should therefore move 10 into another working register, and add it to **temp** using the **add** instruction. Alternatively, you could use ZL, and the **adiw** instruction. You should then repeat the same steps as in **Turning** to send the appropriate message back to the PC. Similarly, the **SlowDown** section subtracts 10 from OCR1AL, forcing the value to 0 if it goes negative. The usual method is used to send the reply to the PC.

The **GoStop** section is slightly harder. You must first test the state of the PWM (i.e. is it enabled?) by testing bit 0 of **TCCR1A**. If it is enabled, disable it, and send 'STOP!' to the PC. If it is enabled, jump to a different section called **Go**. This section should enable 8-bit PWM (set bit 0 of **TCCR1A**), and send 'GO!' to the PC.

The **ChangeSpeed** section has to wait for two more characters (the two digits of the speed). It should start with a loop to wait for the first character (waiting for the RXC bit in **USR** to set). The first digit received should be moved from the **UDR** into a working register called **speed10**. This number should be copied into a temporary register, and have 0x30 subtracted from it. This converts the ASCII for 0–9, into the numbers 0 to 9. The result of this should then be multiplied by 10, as this is the tens digit. The next digit should then be received, and the result stored in a register called **speed1**. Again, convert this into the actual number (subtract 0x30), and add it to the tens digit. It is important you keep **speed10** and **speed1** unchanged, as these will be used when replying to the PC. The value representing the total two-digit number will be between 0 and 99. We would like to convert this to something between 0 and 255 – an easy way to do this is to multiply it by 3, but cap anything that goes above 255. The result should be moved into **OCR1AL**. The reply should be sent to the PC 'Speed Set To xx', with xx being the new two-digit speed. For letters, we move the ASCII values into **temp** as before. For the actual speed, just copy **speed10** or **speed1** into **temp**, and call **Send**, as before. After sending **speedl**, this section should jump to **EndMessage**.

Finally, the hardest section is **Message**. This converts input characters from ASCII into seven segment code, and scrolls the result through the displays as they come in. The display registers will be called **Thousands**, **Hundreds**, **Tens** and **Ones**. As new numbers come in, **Hundreds** will be copied to **Thousands**, **Tens** to **Hundreds**, **Ones** to **Tens**, and finally the new number will be written to **Ones**. First, however, we must convert ASCII to seven segment numbers. We will try to display the digits '0' to '9' only, the lower case letters 'a' to 'z', and the upper case letters 'A' to 'Z', with the exclusions we noted earlier. With the letters, where a lower case letter is not possible whilst an upper case is (e.g. 'e' and 'E'), the upper case alternative is returned. This ensures that the program will try to produce the intended case, but gives getting the letter right at all a higher priority. As you may have guessed, this conversion process is carried out with one large look-up table. The first task is simply to reply to the PC with the character just received. This is straightforward – read **UDR** into **ZL**, disable received mode, enable transmit mode, copy **ZL** into **temp**, and then call the **Send** subroutine. Change back into receive mode and disable transmit mode, and then subtract 0x10 from **ZL**. The digits 0–9 start at 0x30 in ASCII, so subtracting 0x10 will make a '0' correspond to 0x20 etc. This is a byte address, so the word address will be half of this, i.e. a '0' corresponds to word address 0x10. We can make this the start of our look-up table (use **.org 0x10** at the start of the table). The first five words in the look-up table can represent the digits 0–9. Make sure you work out your own values for the look-up table, instead of copying those in my program, as your circuit board may not be the same as mine. Capital letters 'A' to 'Z' start at ASCII value 0x41. Rather than writing empty lines into the look-up table, simply write **.org 0x18**, to point the next part of the look-up table at program address 0x18, which is byte address 0x30, which corresponds to ASCII 0x40. The first byte in the table is therefore not important, but the second should correspond to 'A', and so on. Finally, letters 'a' to 'z' begin at ASCII value 0x61, and so use **.org 0x28** at the top of the look-up table for the lower case letters.

I realized when testing that a space (i.e. pressing the space bar) was an important symbol to transmit. This is 0x20 in ASCII, which gets reduced to byte address 0x10, and word address 0x08. A clever way to deal with spaces, therefore, is to make address 0x08 a **nop** instruction (**nop** is translated as 0x0000 by the assembler). **nop** would be read as any of the other bytes, and return 0b00000000 which corresponds to all bits off (i.e. a space). 0x08 happens to be the UART Empty interrupt, which we are not using, so it is fine to simply write **nop**. In the unforeseeable event that the UART Empty interrupt *does* occur, all that will happen is that it will execute the **nop**, and then the **reti** instruction which follows at address 0x09. The program is therefore still immune to an unexpected occurrence of the UART Empty interrupt. Once the program memory has been read, and the values in the registers shifted along, the **Message** section is finished.

This concludes the final program, my version is shown in Program P. I hope you do try to build this one, and work on some enhancements to make it more robot-like. It really is a good platform for a variety of interesting projects.

Conclusions

When you are debugging your own programs, I suggest the following. First, try to break down your program into discrete units which can be tested independently – this way you can pinpoint bugs quickly. Another frustrating problem can be not being able to look inside the register of the AVR while it is running. This can be overcome by using an emulator, though there is a cheaper way. At certain points in the program you could try sending the contents of certain registers through the UART to your PC, and see how they are changing. The insertion of a UART transmission module in your program may not be worth the extra work, but it does give you a good indication of what's going on *inside* your AVR – like a poor man's JTAG or emulator.

Throughout this book we have encountered examples of attempting to perform a task with limited means, and then learning about new tools which allow us to perform these tasks with greater ease. It is often the case that the more complicated the microcontroller becomes, the simpler a given program will become. This gives us some insight into the compromise that chip designers face between giving a chip functionality and keeping it relatively simple. This simplicity is necessary not only to keep costs low, but also to make the chip easy to get to grips with. I have no doubt that new features will emerge on new models of AVR that appear after the publication of this book. These will almost inevitably centre around some I/O register, perhaps with a certain bit assignment that controls different aspects. This information can be gleaned from the chip's datasheets, which should not be as daunting now as they might have been when you started. By reading through these you should be able to keep abreast of any new functions – make sure you keep up to date with these, they're there to make your life as a programmer easier!